UM EECS 270 F22
Introduction to Logic Design

17. Sequential Multiplication

Multiplication

- Combinational
  - Parallel
  - Fast
  - High area cost
- Sequential
  - Serial
  - Slow (*n* cycles for *n*-bit numbers)
  - Reasonable area cost

multiplication is actually repeated addition

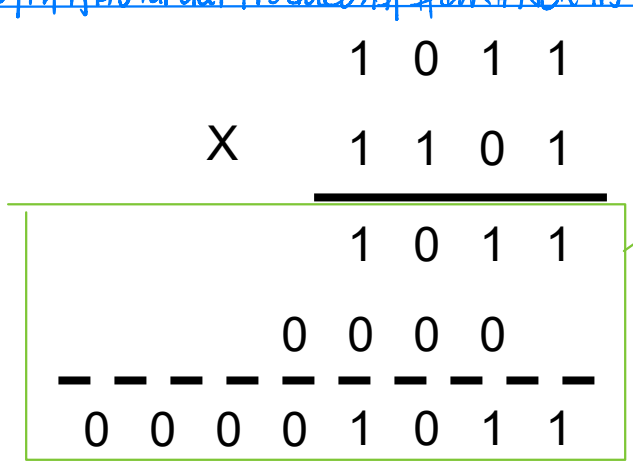
Multiplication Example

				1	0	1	1		Multiplicand Q (11_{10})
	X			1	1	0	1		Multiplier M (13_{10})
				1	0	1	1		Partial Product P_0 (11_{10})
			0	0	0	0			Partial Product P_1 (0_{10})
		1	0	1	1				Partial Product P_2 (44_{10})
	1	0	1	1					Partial Product P_3 (88_{10})
1	0	0	0	1	1	1	1		Product P (143_{10})

不同于addition, Sum只须保存和operands相同的bits数.
除有可能出现overflow bit, 对于multiplication, 仔细让
multiplication的result至少2n bits (n bits x n bits的结果

Multiplication Example

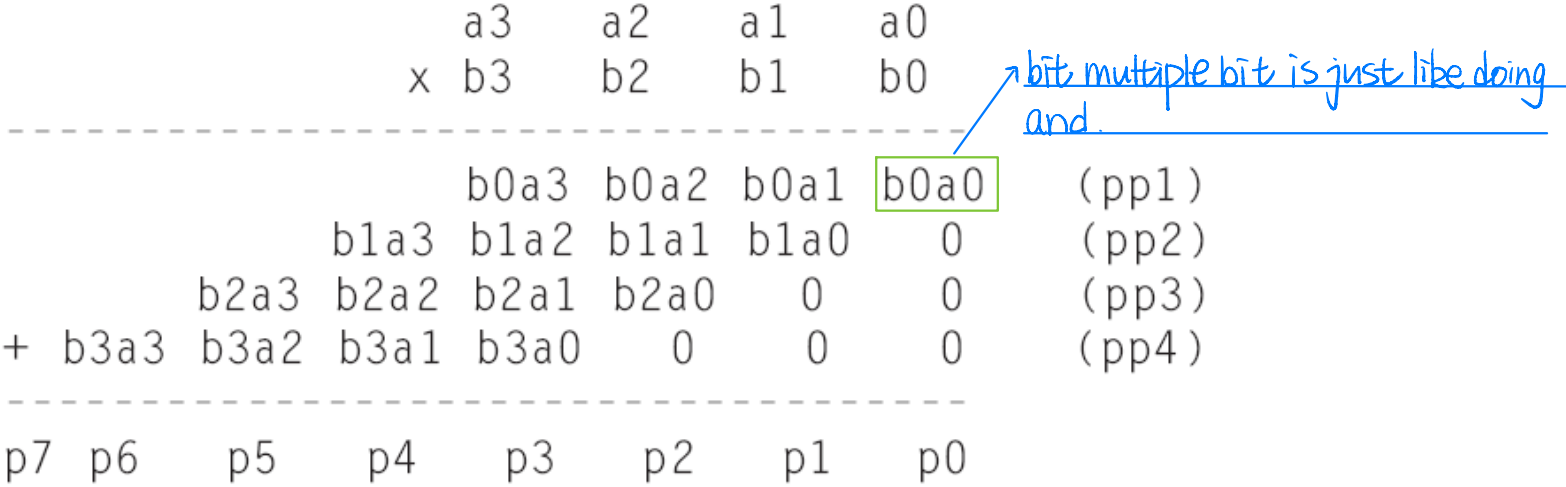
相比较上一页multiplication方法有改进：
与其等到所有的Partial Product都算出来，再把他们加合在一起，我们可以边算边加。



				1	0	1	1		Multiplicand Q (11_{10})
		X		1	1	0	1		Multiplier M (13_{10})
				1	0	1	1		Partial Product P_0 (11_{10})
			0	0	0	0			Partial Product P_1 (0_{10})
0	0	0	0	1	0	1	1		Accumulated Sum (11_{10})
		1	0	1	1				Partial Product P_2 (44_{10})
0	0	1	1	0	1	1	1		Accumulated Sum (55_{10})
	1	0	1	1					Partial Product P_3 (88_{10})
1	0	0	0	1	1	1	1		Product P (143_{10})

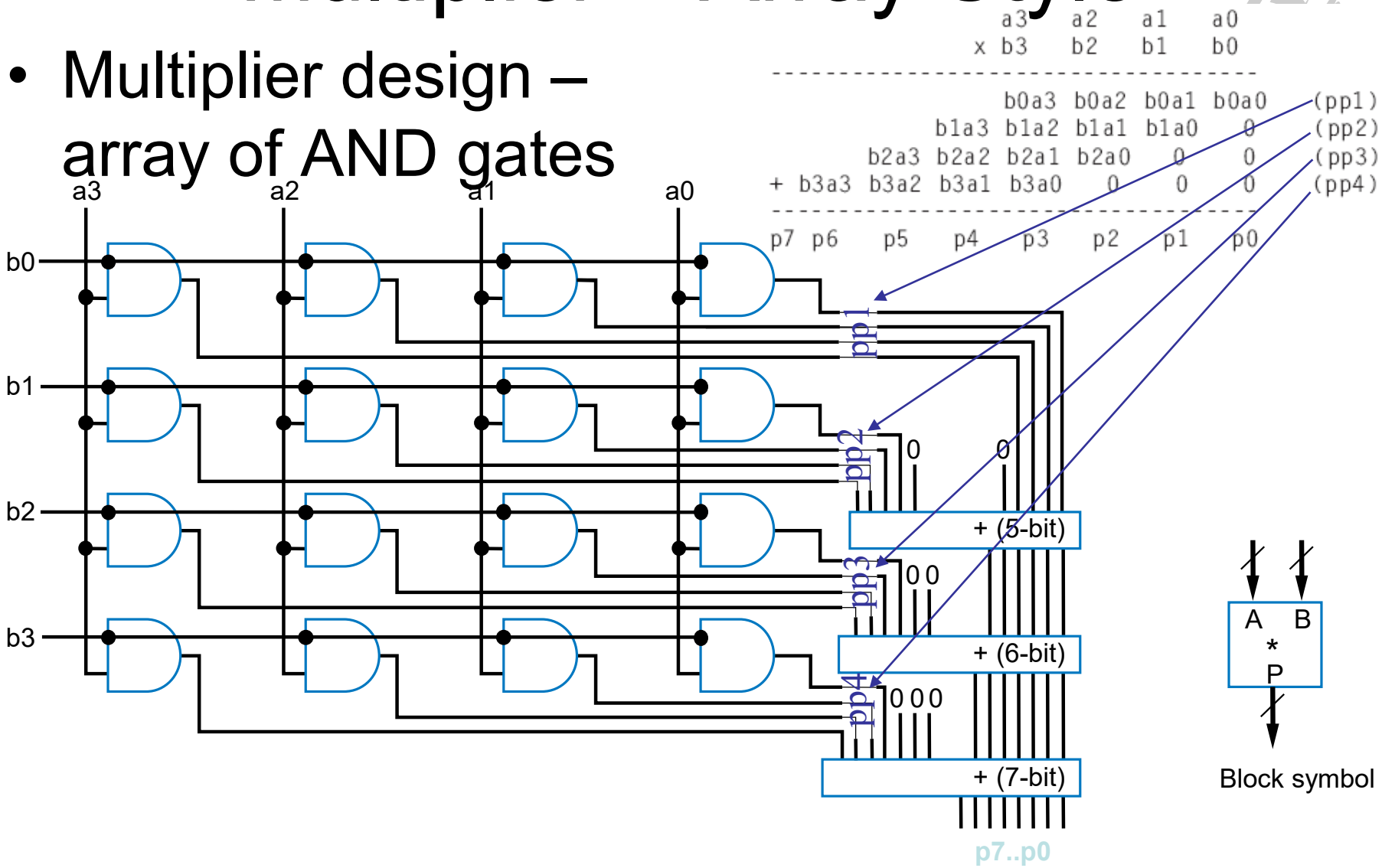
Multiplier – Array Style

- Generalized representation of multiplication by hand



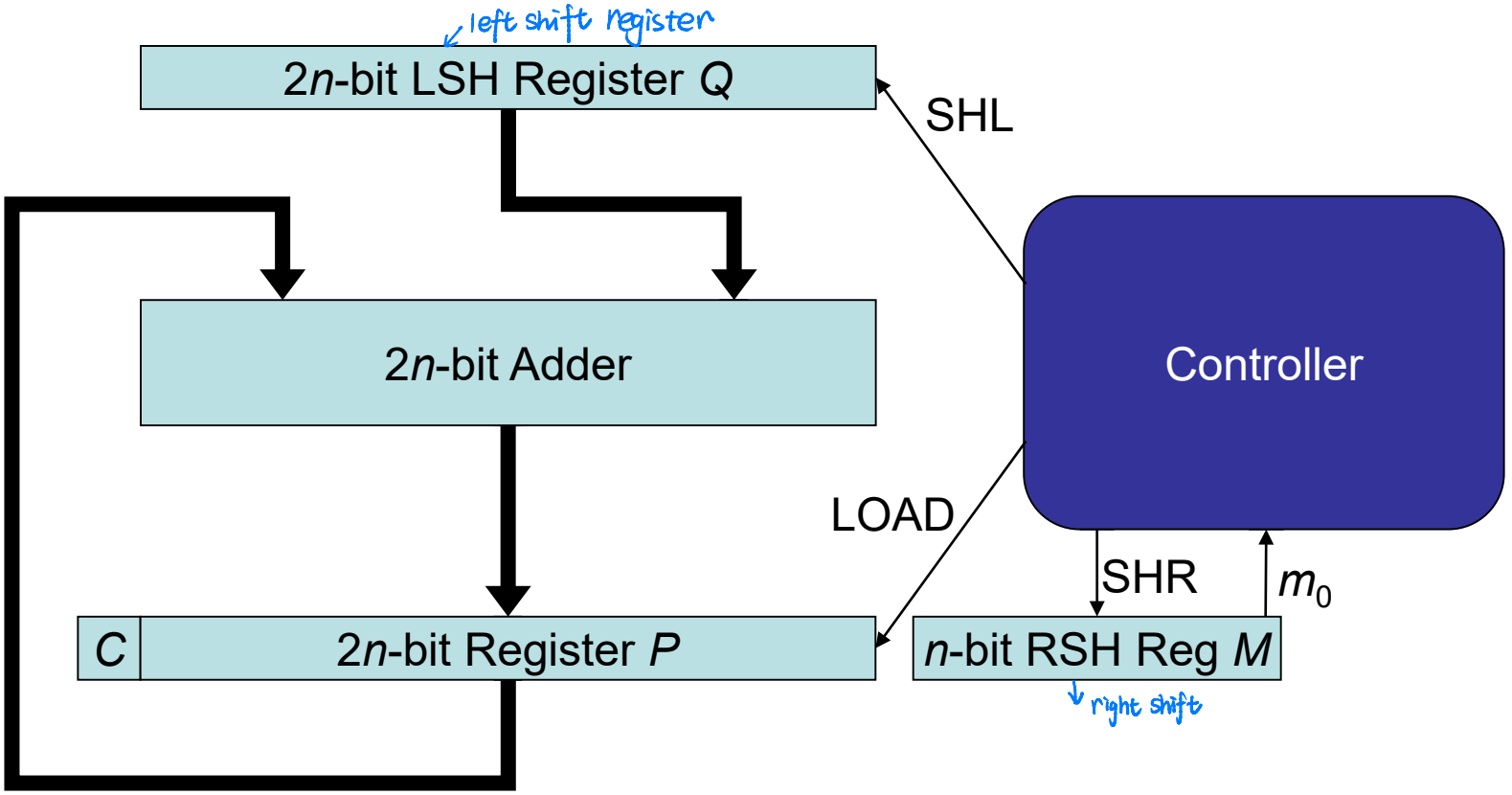
Multiplier – Array Style

- Multiplier design – array of AND gates



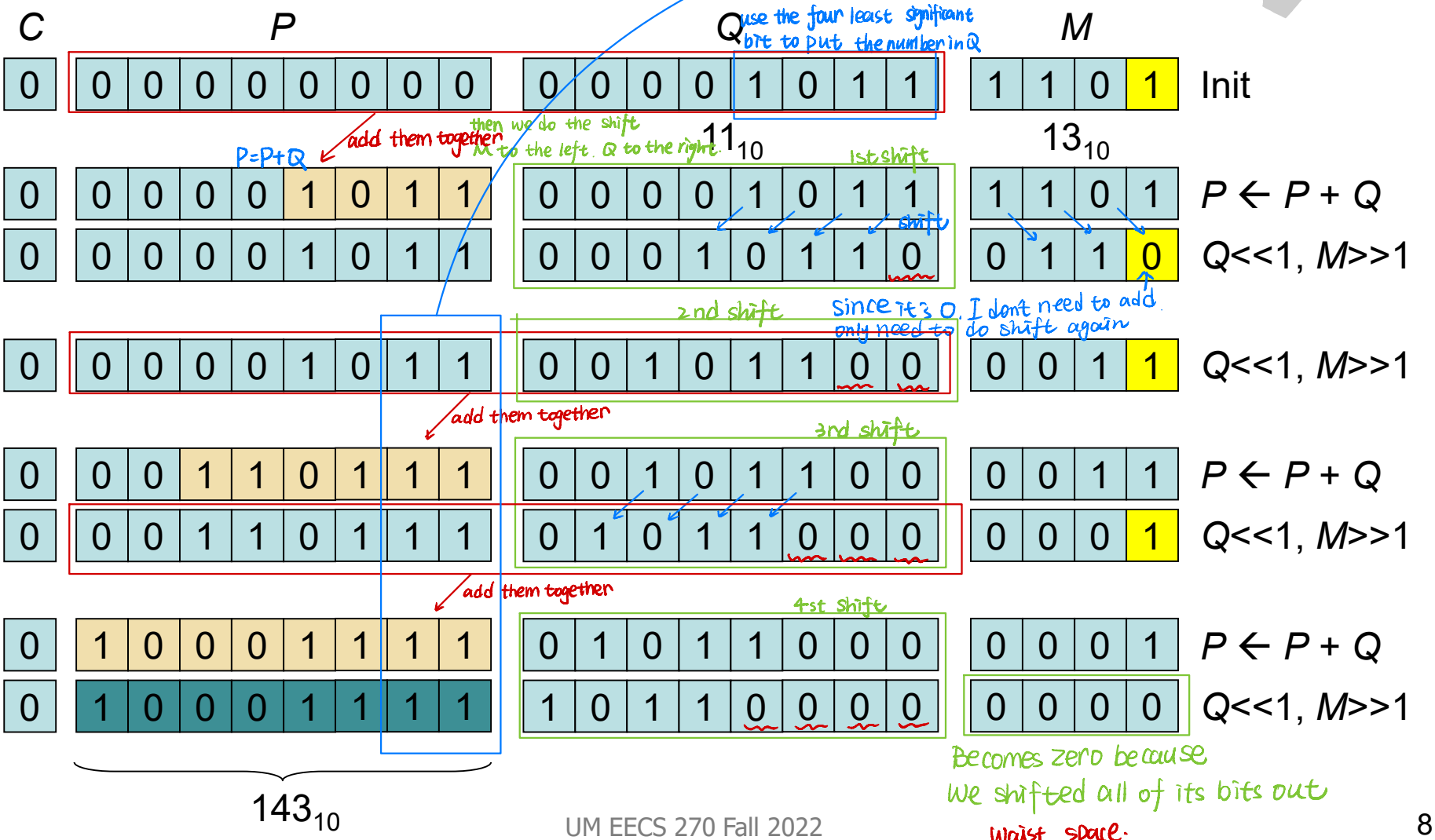
Sequential Multiplier V0.1

we need a 2n wide data path. the reason is the final results is 2-n bits wide.



Multiplier V0.1 Execution Trace

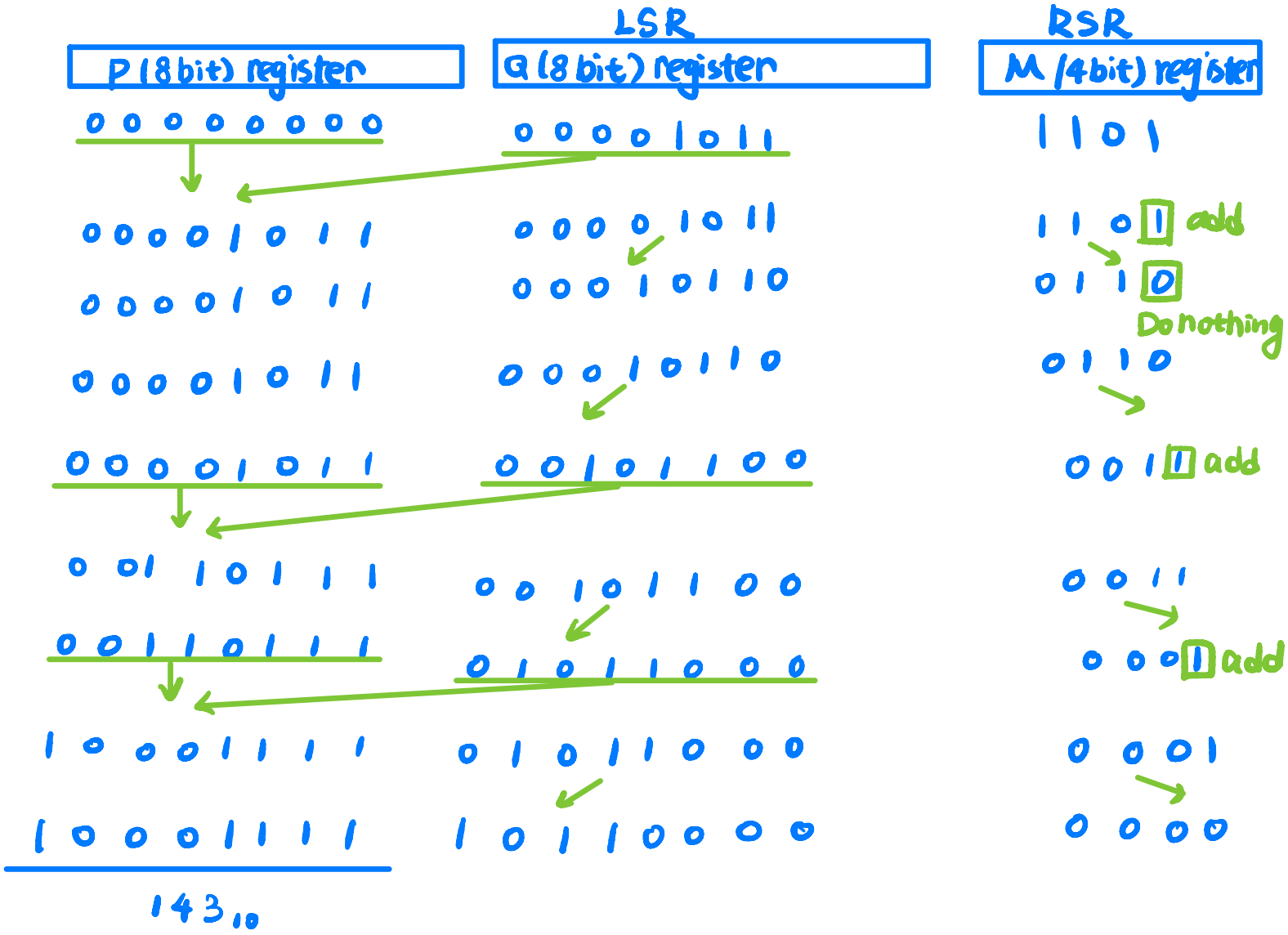
inefficient：因为在shift Q to the flet的过程中，its least significant bits are filled with zeros
So we use 2n bits adder，但因为Q最右侧是0，则P最右侧在加的过程中没有发生变化 → 2n bits adder is overkilled.

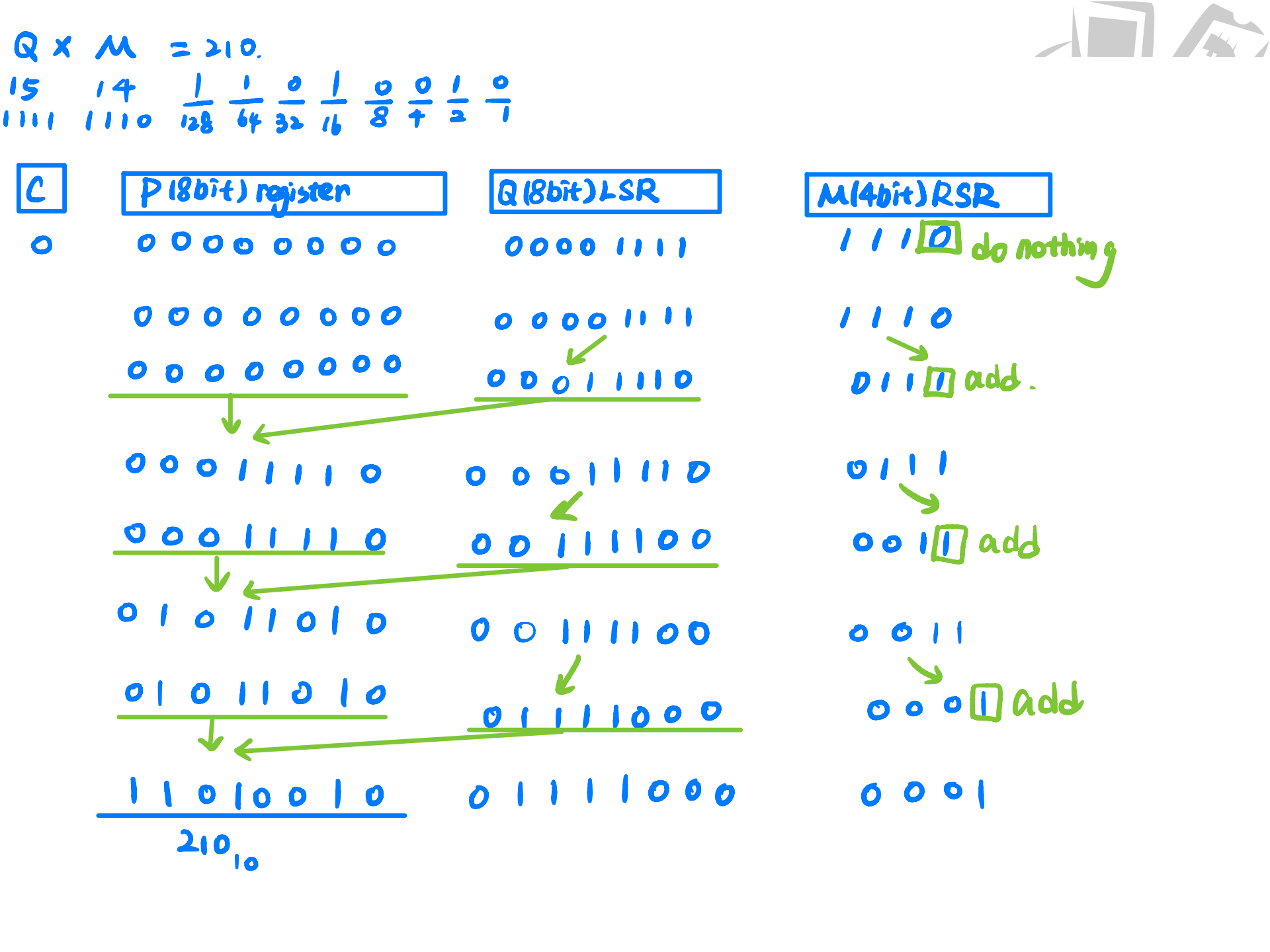


Q x M = 143_{10}

1011 1101

1.10 13_{10}




Q x M = 210.
15 14 1 1 0 1 0 0 1 0
1111 1110 128 64 32 16 8 4 2 1
C
P (8bit) register
Q (8bit) LSR
M (4bit) RSR
0
0000 0000
0000 1111
1110 do nothing
0000 0000
0000 1111
1110
0000 0000
0001 1110
1111 add.
0001 1110
0001 1110
0111
0010 1101
0011 1100
001 add
0010 1101
0011 1100
011
0010 1101
0111 1000
000 add
1101 0010
0111 1000
0001
210 10

Sequential Multiplier V1.0

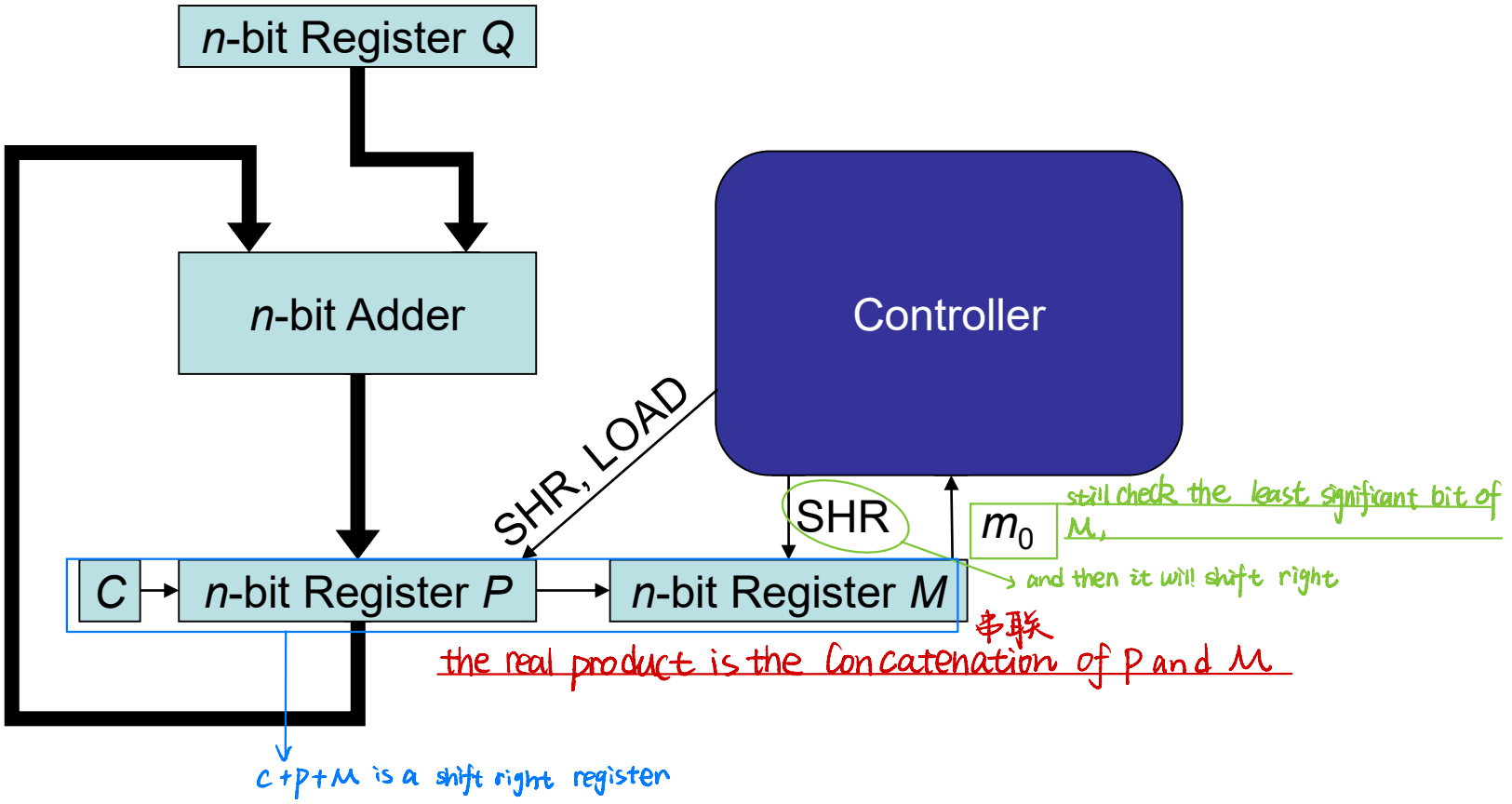
- 2*n*-bit adder is overkill since least significant bits of product don't change after they are initially formed (they are added to zeros!)
  - *n*-bit adder is sufficient
- Left shifting the multiplicand causes half the space in the 2*n*-bit *Q* and *P* registers to be unutilized 刚开始Q左侧n bit unutilized,
  我们waist space by using a 2n bits adder and using these 2n bits registers.

解决方法:
  - Use an *n*-bit register for *Q* (no shifting)
  - Use a combined (2*n*+1)-bit *C*+*P*+*M* right-shift register

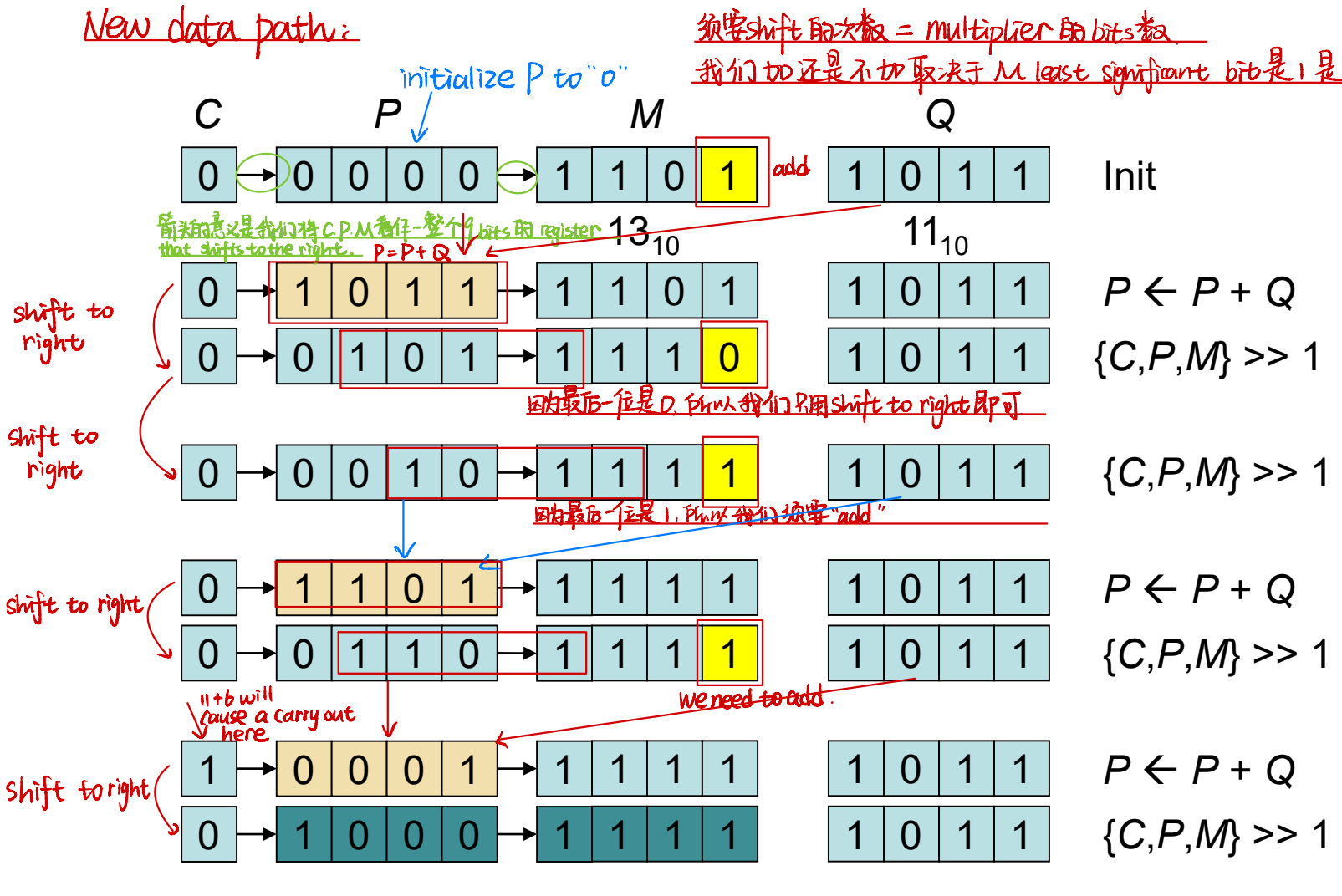
carry out

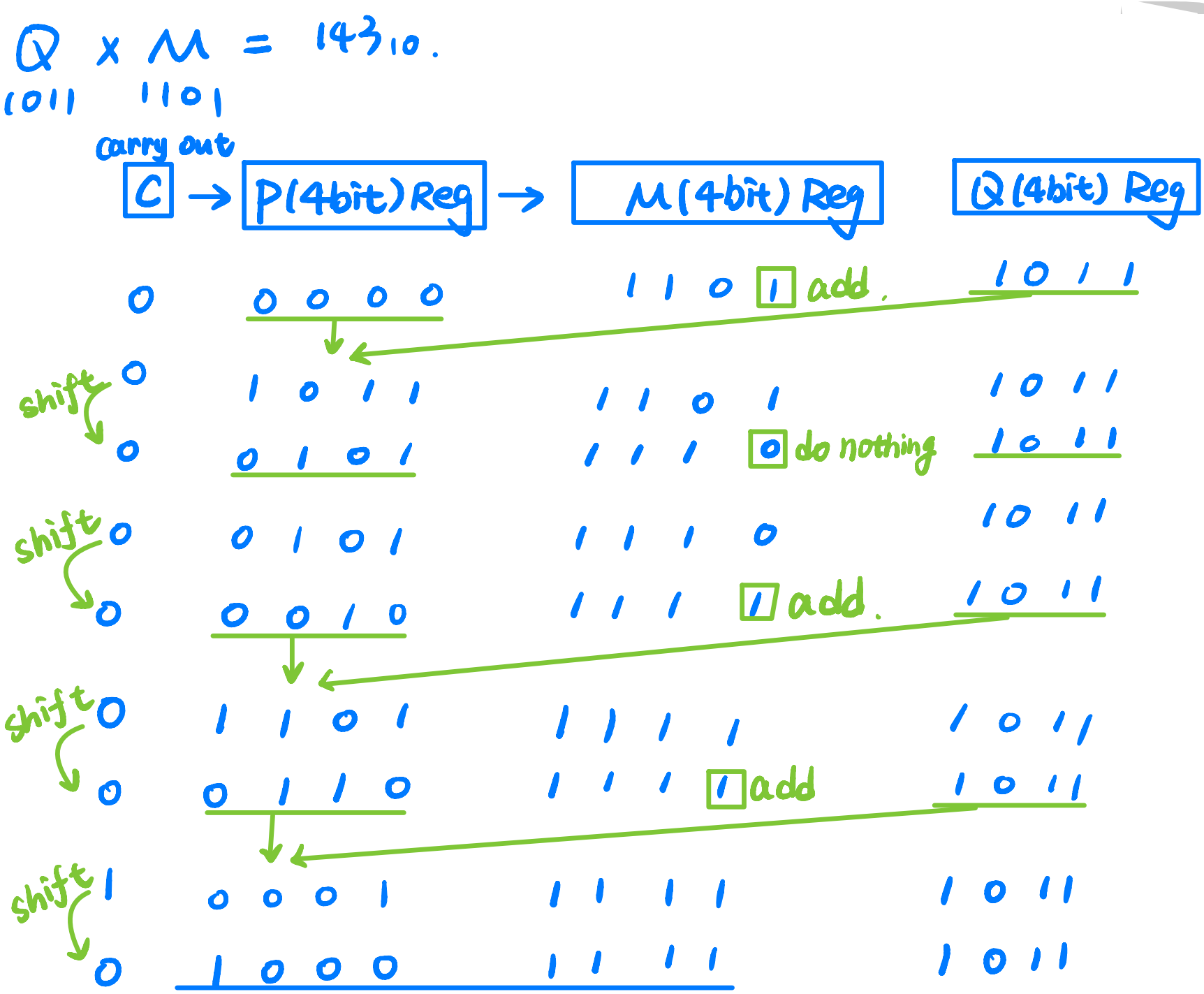
each of them is n-bits register

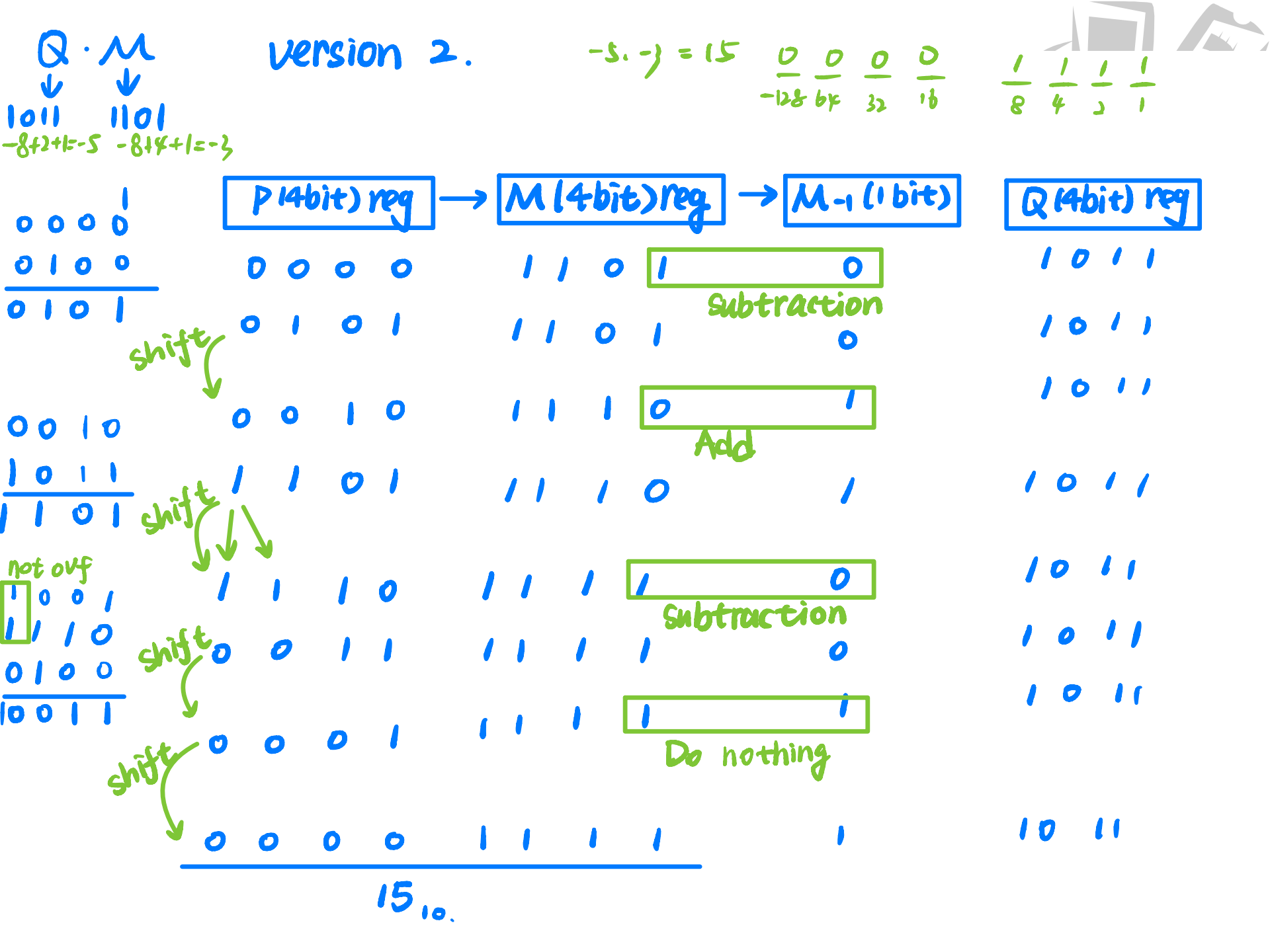
Sequential Multiplier V1.0



Multiplier V1.0 Execution Trace




Q x M = 14310.
1011 1101
carry out
C
P(4bit) Reg
M(4bit) Reg
Q(4bit) Reg
0 0000 1101 1011
add.
shift
0 1011 1101 1011
0 0101 1111 1011
do nothing
shift
0 0101 1111 1011
0 0010 1111 1011
add.
shift
0 1101 1111 1011
0 0110 1111 1011
add
shift
1 0001 1111 1011
0 1000 1111 1011


Q · M
version 2.
-5 · -3 = 15
1011
1101
-8+2+1=-5
-8+4+1=-3
P (4bit) reg
M (4bit) reg
M-1 (1 bit)
Q (4bit) reg
Subtraction
shift
Add
shift
shift
not ovf
Subtraction
shift
Do nothing
shift
15 10.

Multiplying Negative Numbers

- This does not work!
- Solution 1
  - Convert to positive if required
  - Multiply as above
  - If signs were different, negate answer
- Solution 2 better
  - Booth's algorithm

Observation

- Which of these two multiplications is more difficult?

  98,765 x 10,001

  98,765 x 9,999

  same complexity.

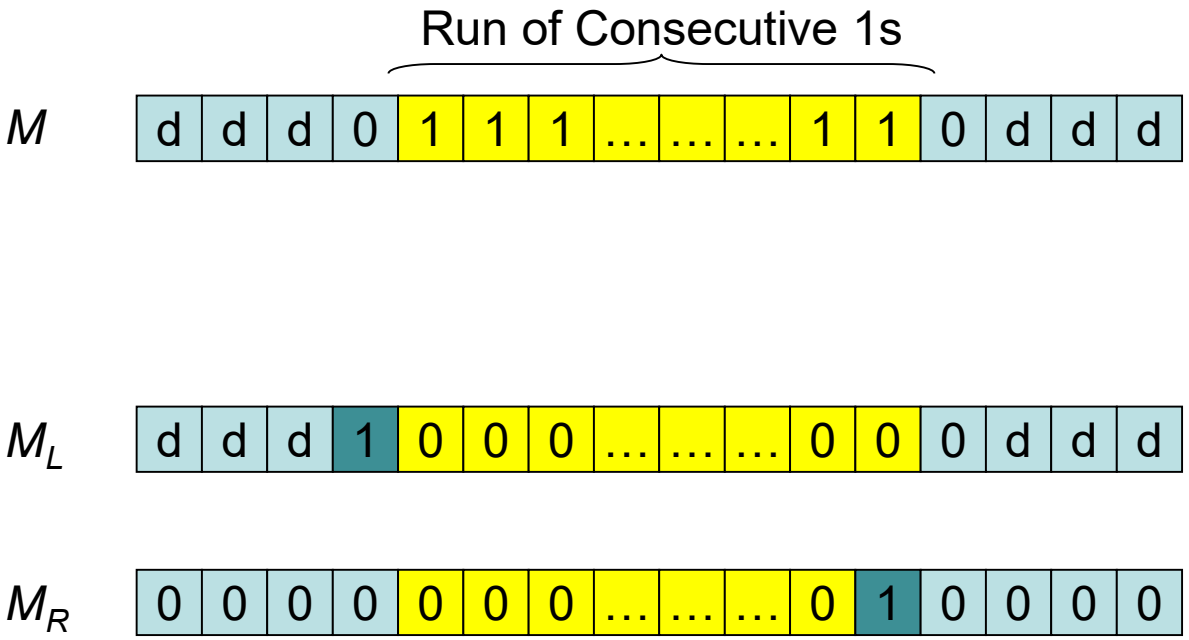
- Note:

  98,765 x 10,001 = 98,765 x (10,000 + 1)

  98,765 x 9,999 = 98,765 x (10,000 – 1)

In Binary

In Binary, the equivalent of 9 (the largest digit) is "1"



$$M = M_L - M_R$$

Let's do the math

Let $P = Q \times M$

where Q and M are n-bit two's complement integers

e.g., $M = -2^3 m_3 + 2^2 m_2 + 2^1 m_1 + 2^0 m_0$

which can be re-written as

$$M = -2^3 m_3 + 2^2 (2-1) m_2 + 2^1 (2-1) m_1 + 2^0 (2-1) m_0$$

$$= -2^3 m_3 + (2^3 - 2^2) m_2 + (2^2 - 2^1) m_1 + (2^1 - 2^0) m_0$$

$$= 2^3 (m_2 - m_3) + 2^2 (m_1 - m_2) + 2^1 (m_0 - m_1) - 2^0 m_0$$

$$= 2^3 (m_2 - m_3) + 2^2 (m_1 - m_2) + 2^1 (m_0 - m_1) + 2^0 (m_{-1} - m_0)$$

$m_{-1} = 0$

yielding $P = Q \times 2^0 (m_{-1} - m_0) + Q \times 2^1 (m_0 - m_1) + Q \times 2^2 (m_1 - m_2) + Q \times 2^3 (m_2 - m_3)$

And here comes the great result!

$$P = Q \times 2^0 (m_{-1} - m_0) +$$
$$Q \times 2^1 (m_0 - m_1) +$$
$$Q \times 2^2 (m_1 - m_2) +$$
$$Q \times 2^3 (m_2 - m_3)$$

now we look at ajacent bits of m. depending on their combinations.

unlike for the unsigned case, where we always either have ① don't do anything ② add.

Now, for the signed number we can do three things: ① don't do anything ② add ③ subtract.

这取决于 Two bits of m at any given time.

m_i	m_{i-1}	$m_{i-1} - m_i$	Action
0	0	0	NOP
0	1	1	Add Q
1	0	-1	Subtract Q
1	1	0	NOP

if they are equal, we don't do anything

Unsigned vs. Signed Multiplication

$$P = Q \times M$$

Unsigned

$$M = 2^3 m_3 + 2^2 m_2 + 2^1 m_1 + 2^0 m_0$$

$$P = Q \times 2^0 m_0 +$$
$$Q \times 2^1 m_1 +$$
$$Q \times 2^2 m_2 +$$
$$Q \times 2^3 m_3$$

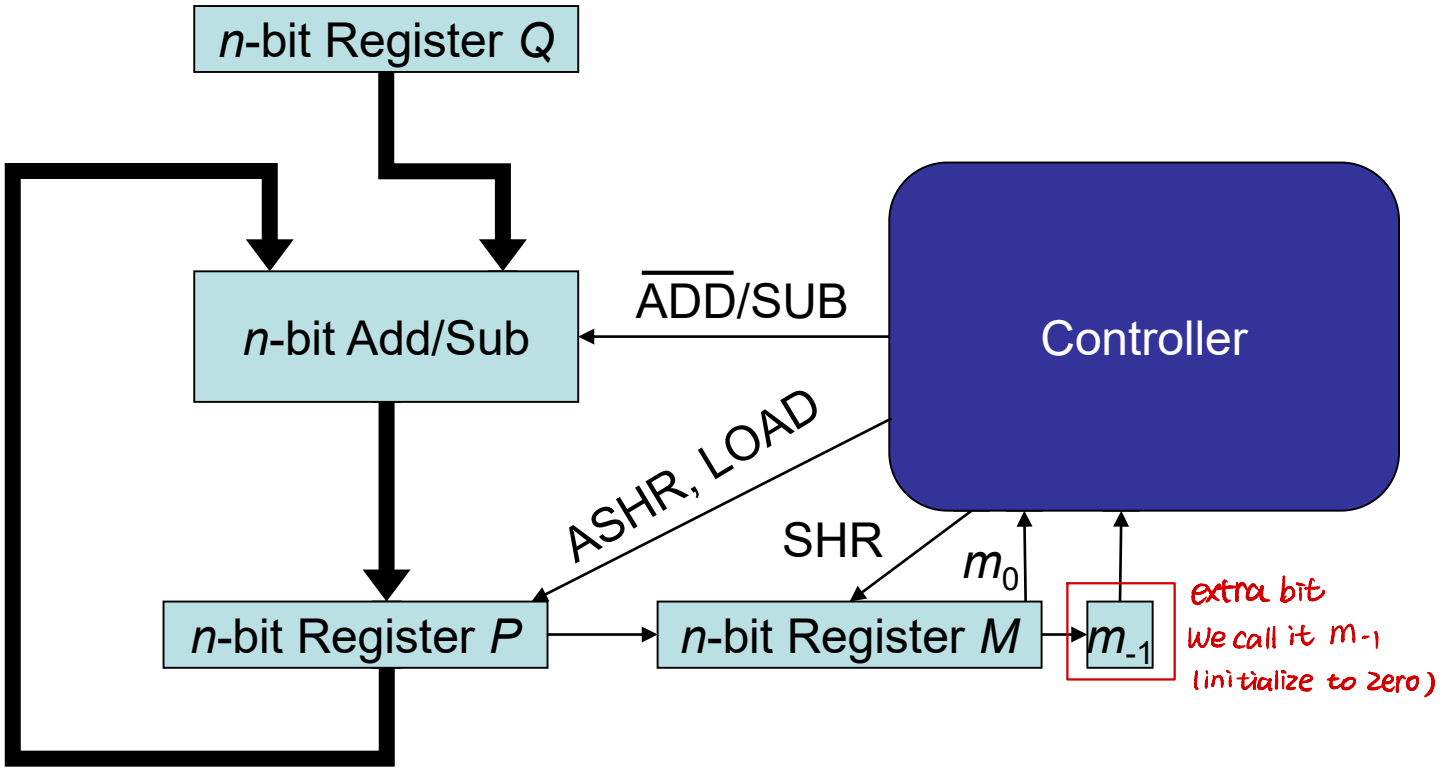
Signed

$$M = -2^3 m_3 + 2^2 m_2 + 2^1 m_1 + 2^0 m_0$$

$m_{-1} = 0$

$$P = Q \times 2^0 \left(m_{-1} - m_0\right) +$$
$$Q \times 2^1 \left(m_0 - m_1\right) +$$
$$Q \times 2^2 \left(m_1 - m_2\right) +$$
$$Q \times 2^3 \left(m_2 - m_3\right)$$

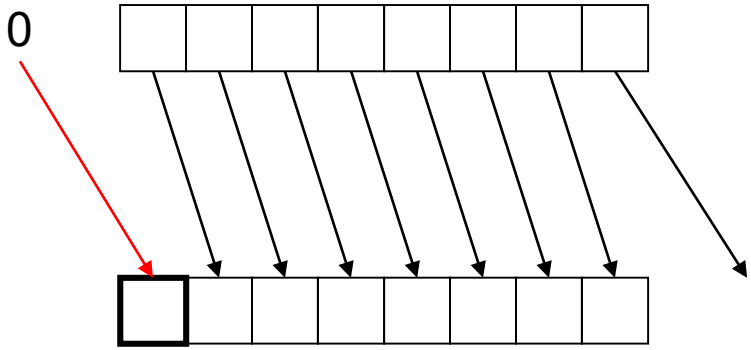
Sequential Multiplier V2.0



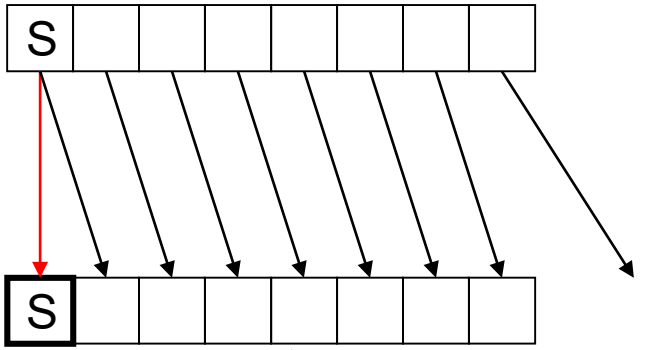
the other thing we need to do: Shifting how is arithmetic shifting.

之前的shift是整体向右移，然后在最高位补"0"，但如果这是一个negative number，shift完还变成了positive，就不对了

Logical vs. Arithmetic Right Shift



Logical Shift

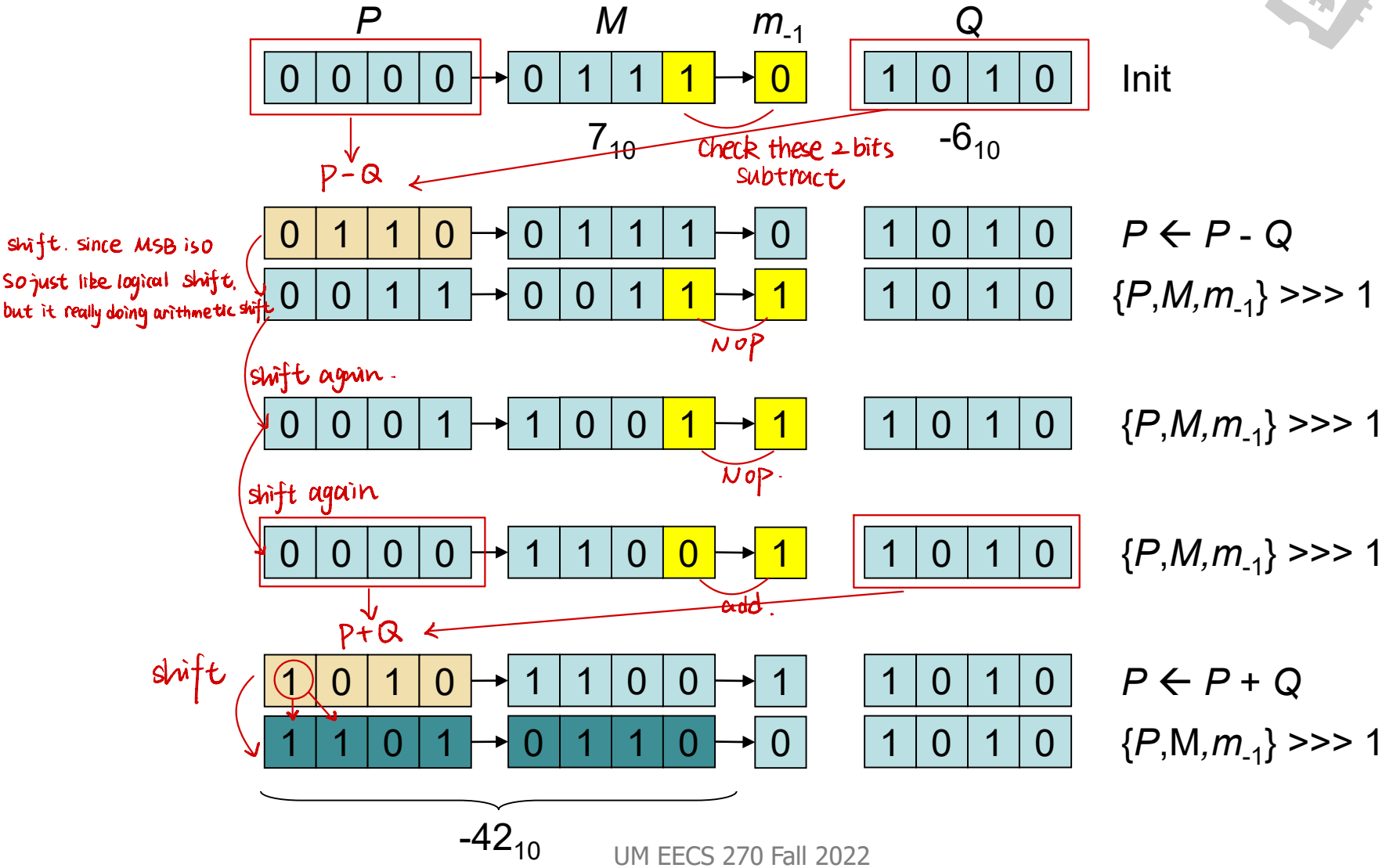


the sign of the number doesn't change, but the shift here is actually divided by 2.

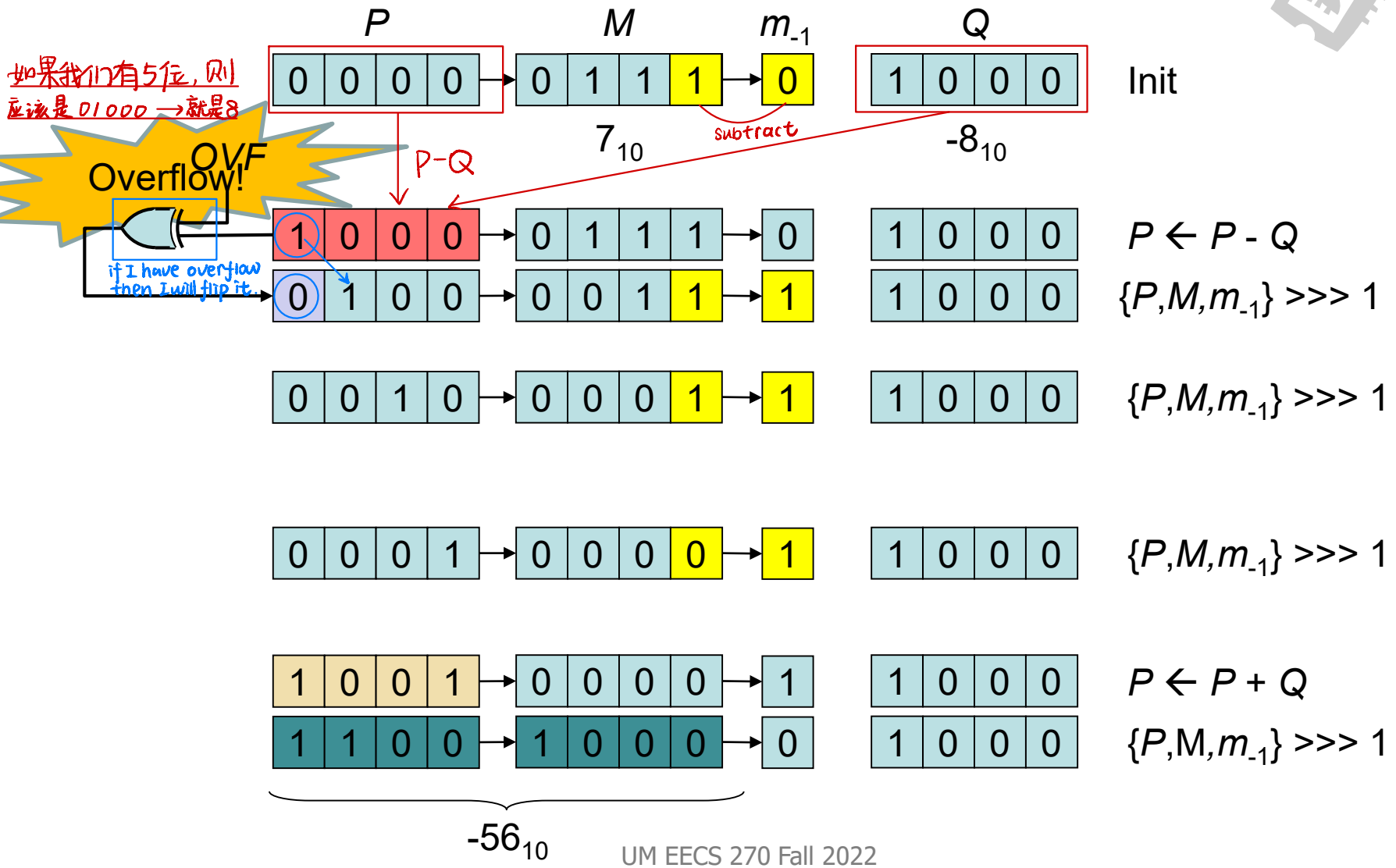
Arithmetic Shift

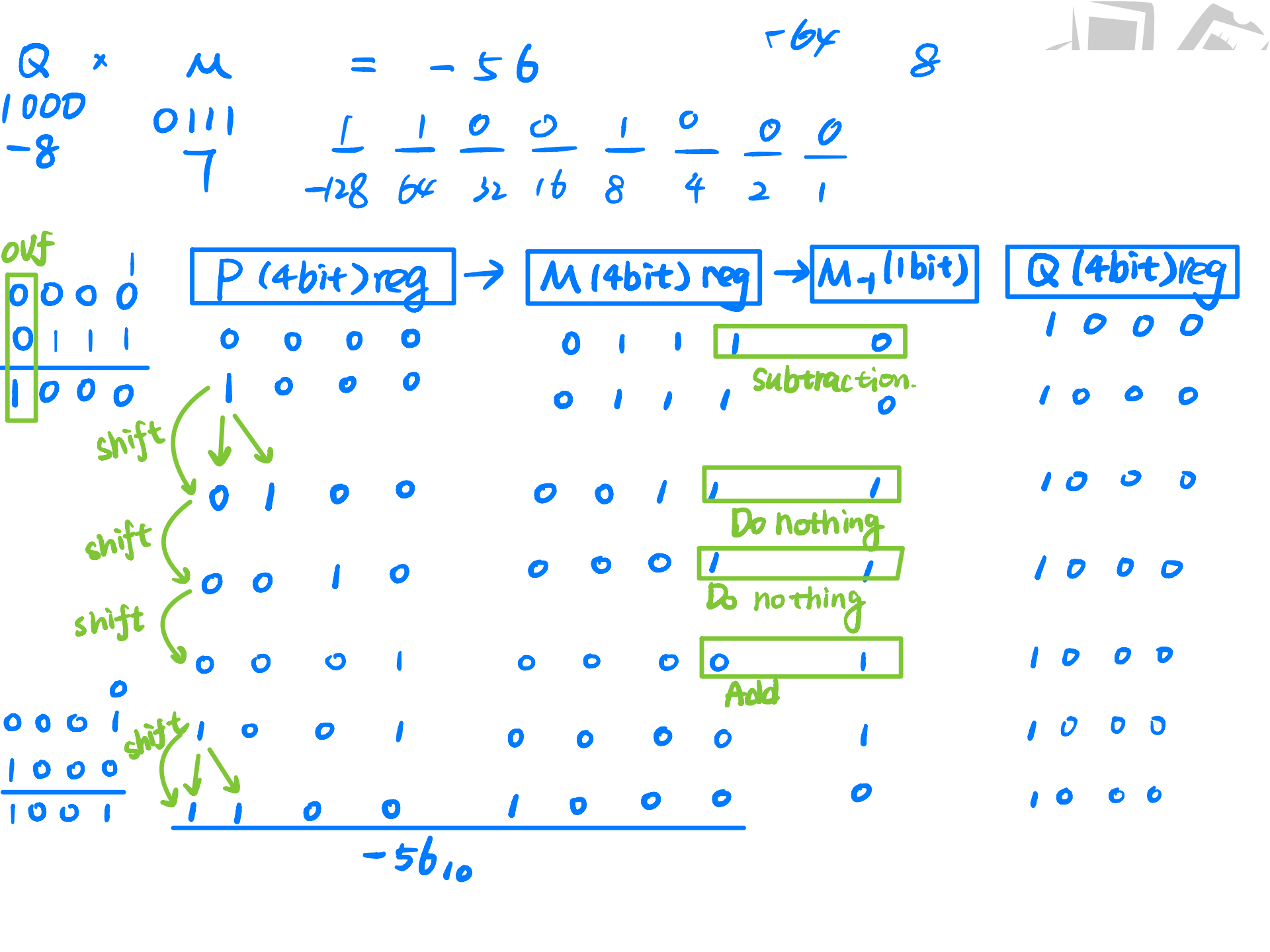
Multiplier V2.0 Execution Trace

the range of 4bit 2s complement number is [-8, 7]

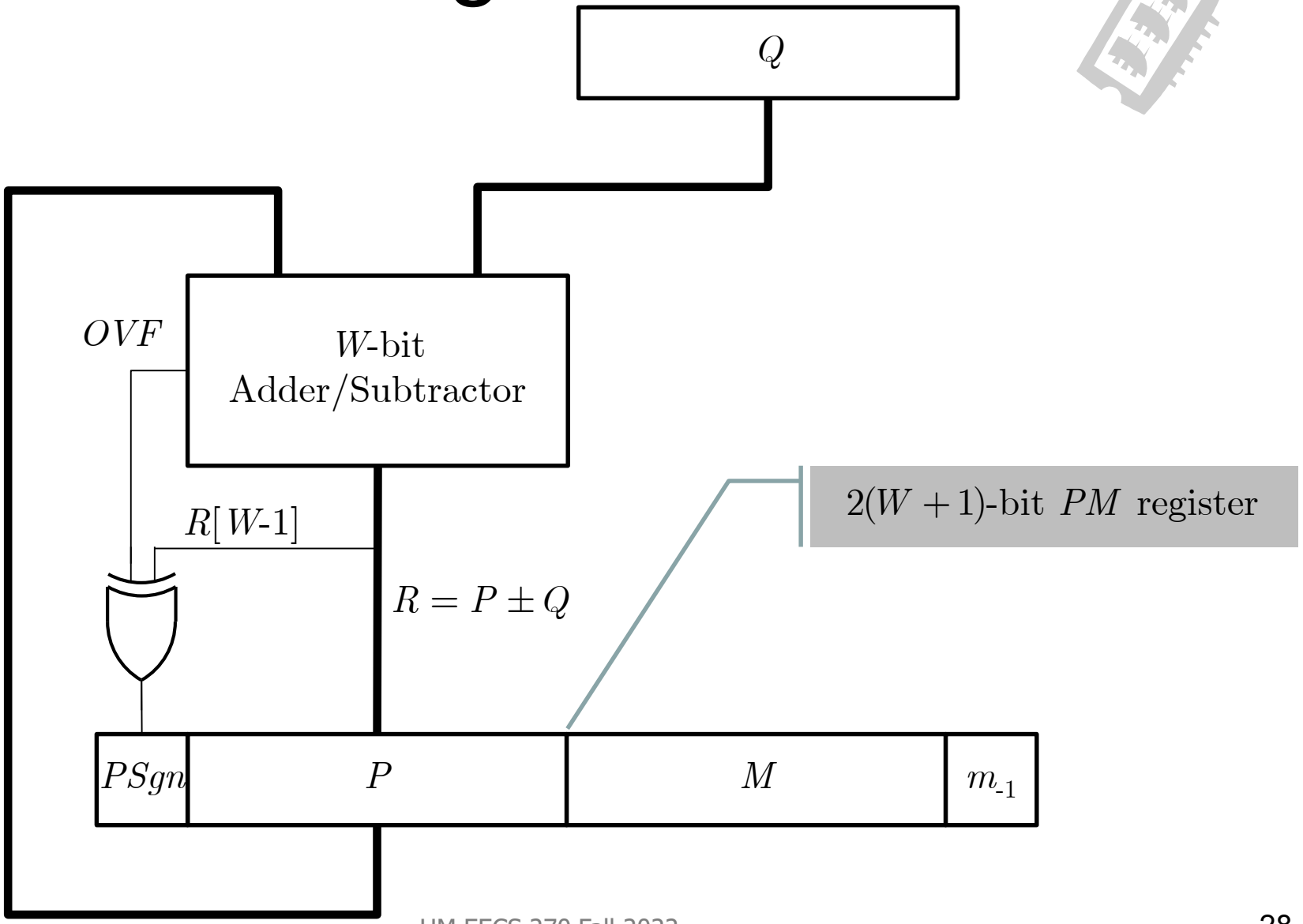


Multiplier V2.0 Execution Trace

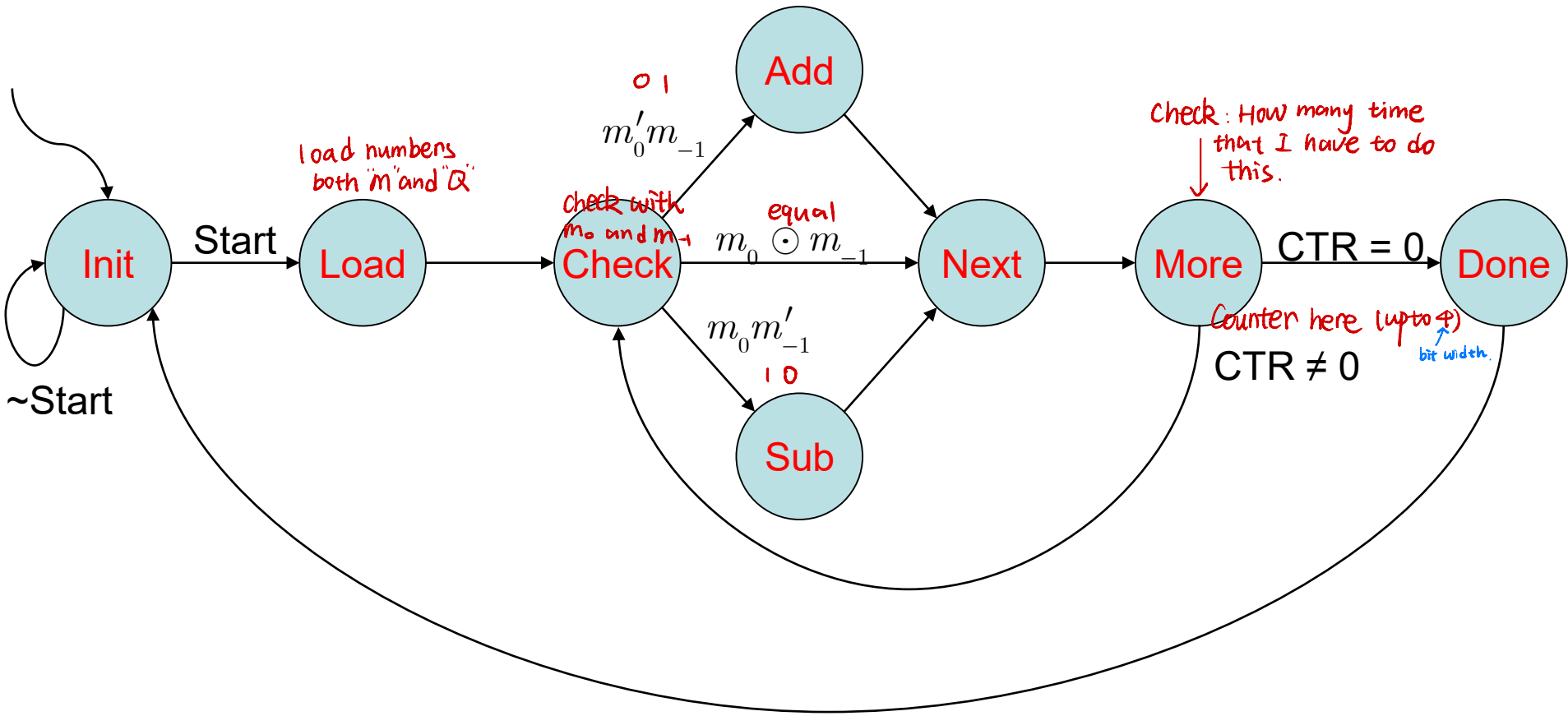



Q × M = −56
1000
−8
0111
7
1 1 0 0 1 0 0 0
−128 64 32 16 8 4 2 1
ovf
P (4bit) reg
M (4bit) reg
M−1 (1bit)
Q (4bit) reg
shift
shift
shift
shift
Subtraction.
Do nothing
Do nothing
Add
−56₁₀

Correcting for Overflow



Controller State Diagram



```
module BoothMul
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module BoothMul
 #(parameter W = 16)
 (Clock, Reset, Start, M, Q, P);
 localparam WW = 2 * W;
 localparam BoothIter = $clog2(W);
 input Clock;
 input Reset;
 input Start;
 input signed [W-1:0] Q;
 input signed [W-1:0] M;
 output signed [WW-1:0] P;
```

ceiling of $\log_2$(width).
it tells you how wide a counter you need in order to count W.

show its 2's complement

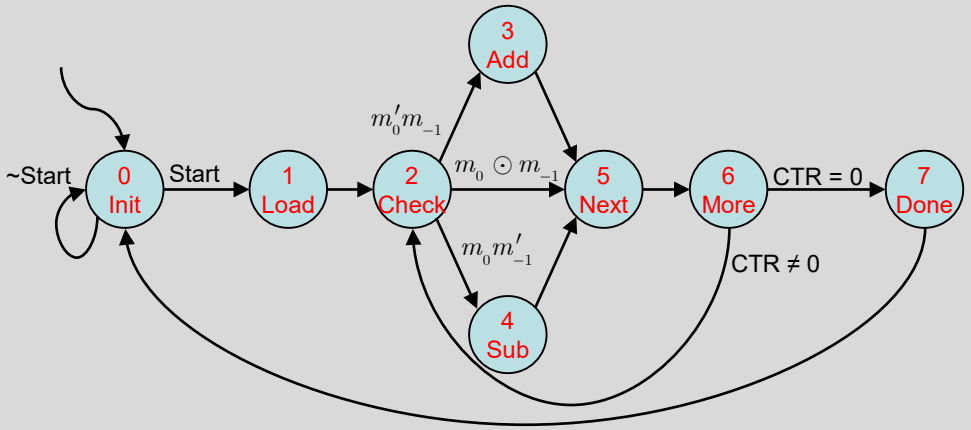
```
endmodule
```

```
module BoothMul
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module BoothMul
 reg signed [WW+1:0] PM;
 reg [BoothIter-1:0] CTR;

 wire c0;
 wire ovf;
 wire signed [W-1:0] R;
 AddSub #(.W(W))
   AddSub1(PM[WW:W+1], Q, c0, R, ovf);
 wire PSgn = R[W-1] ^ ovf;




endmodule
```

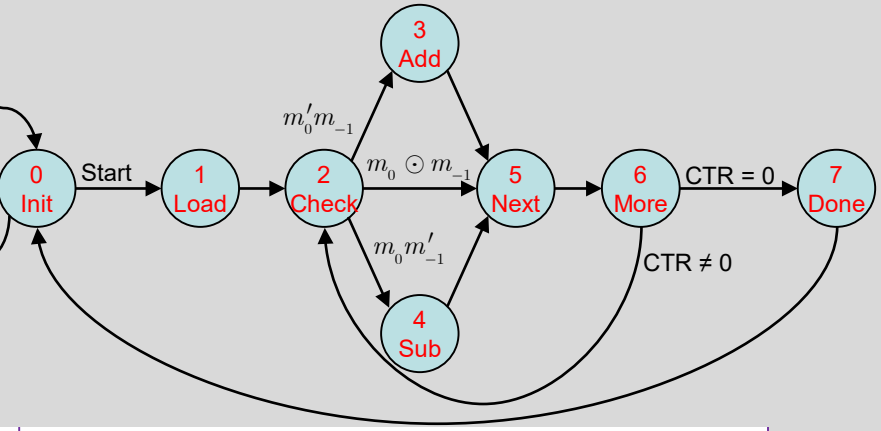
```
module BoothMul
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module BoothMul
 reg [2:0] State, State_Next;
 localparam Init   = 3'd0;
 localparam Load   = 3'd1;
 localparam Check  = 3'd2;
 localparam Add    = 3'd3;
 localparam Sub    = 3'd4;
 localparam Next   = 3'd5;
 localparam More   = 3'd6;
 localparam Done   = 3'd7;
```




```
endmodule
```

```
module BoothMul
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller Transitions
```




```
// Datapath Output Logic

endmodule
```

```
module BoothMul
 always @*
 begin
   case(State)
     Init:
       if (Start)
         State_Next <= Load;
       else
         State_Next <= Init;
     Load: State_Next <= Check;
     Check:
       if (~PM[1] & PM[0])
           State_Next <= Add;
       else if (PM[1] & ~PM[0])
           State_Next <= Sub;
       else
           State_Next <= Next;
     Add: State_Next <= Next;
     Sub: State_Next <= Next;
     Next: State_Next <= More;
     More:
       if (CTR == 'd0)
           State_Next <= Done;
       else
           State_Next <= Check;
     Done: State_Next <= Init;
   endcase
 end

endmodule
```

```
module BoothMul
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module BoothMul
 initial
 begin
   State <= Init;
   PM <= 'd0;
   CTR <= W;
 end
```

counter: number of bits

```
endmodule
```

```
module BoothMul
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module BoothMul
 always @(posedge Clock)
   if (Reset)
     State <= Init;
   else
     State <= State_Next;



endmodule
```

```
module BoothMul
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module BoothMul
 assign M_LD = (State == Load);
 assign P_LD =
   (State == Add) | (State == Sub);
 assign PM_ASR = (State == Next);
 assign CTR_DN = (State == Next);
 assign AllDone = (State == Done);




endmodule
```

```
module BoothMul
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module BoothMul
 wire signed [W:0] ZERO;
 assign ZERO = 'd0;
 always @(posedge Clock)
   if (Reset)
     begin
       PM[WW+1:W+1] <= 'd0;
       PM[0] <= 0;
       CTR <= W;
     end
   else
     begin
       PM <=
         (M_LD? $signed({ZERO, M, 1'b0}):
           (P_LD? $signed({PSgn, R, PM[W:0]}):
             (PM_ASR? PM >>> 1:
               PM
             )
           )
         );

       CTR <= CTR_DN? CTR - 1 : CTR;
     end
```

You can not update piece of the register, you must update all of them. So the register PM is what we called P+M+m-1

if I want to load W.

if I'm loading P.

shift by 1.

```
module BoothMul
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module BoothMul
 assign P = AllDone? PM[WW:1] : 'd0;


endmodule
```

ModelSim Demo